OCUNS: una arquitectura para la enseñanza de tópicos de Organización y Arquitectura de Computadoras

Pablo M. Davicino*, Alejandro G. Stankevicius, and Rafael B. García

Laboratorio de Investigación de Sistemas Distribuidos (LISiDi),
Departamento de Ciencias e Ingeniería de la Computación
Teléfono: +54 291 4595135, Fax: +54 291 4595136
Universidad Nacional del Sur, Bahía Blanca (8000), Argentina
{pmd,ags,rbg}@cs.uns.edu.ar,
WWW home page: http://lisidi.cs.uns.edu.ar

Resumen Las asignaturas que componen la currícula de ciencias de la computación se clasifican en diversas áreas. Aquellas que conforman el área de sistemas presentan un conjunto de desafíos propios, dado su inherente bajo nivel de abstracción producto de la proximidad al hardware de muchos de sus conceptos fundacionales. Ha sido nuestra experiencia en el dictado de la asignatura *Organizacion de om adoras*, la que nos ha indicado el claro beneficio experimentado por los alumnos al poseer una herramienta que les permita interactuar con el hardware sobre el cual han de desarrollar la práctica. En consonancia con esta tesitura, el presente artículo caracteriza formalmente una arquitectura de juguete denominada OCUNS la cual puede ser utilizada como herramienta para asistir en la enseñanza de diversos tópicos del área de sistemas, argumentando a su vez los beneficios desde un punto de vista tanto pedagógico como metodológico. Por último, también se analizan las principales características de una implementación en JAVA para esta arquitectura.

Keywords: arquitectura de juguete, organización de computadoras, arquitectura de computadoras, aprendizaje por descubrimiento

1. Introducción y motivaciones

Las distintas asignaturas que componen la currícula de ciencias de la computación se clasifican, en ocasiones de una manera no tanto *de jure* sino más bien *de facto*, en diversas área o ramas, en particular si tenemos en cuenta la interrelación entre éstas a nivel de contenidos y de correlatividades. Si bien la demarcación precisa de estas áreas cambia de una universidad a otra, en general un área cubre las *ciencias básicas*, otra la *programación*, otra quizás la *ingeniería de software*, y otra usualmente las nociones y conceptos más cercanos al hardware y al sistema operativo denominada *sistemas* (al menos en nuestra unidad

* Becario de la Comisión de Investigaciones Científicas de la Provincia de Buenos Aires.

académica [1], el Departamento de Ciencias e Ingeniería de la Computación de la Universidad Nacional del Sur). Esta área abarca desde cuestiones básicas como los distintos sistemas de representación numérica, hasta aspectos más avanzados como los mecanismos de sincronización en sistemas distribuidos o los protocolos de las distintas capas de las arquitecturas de red. Este artículo se focaliza en las particularidades que enfrentan los docentes del área de sistemas, en el momento de transmitir ciertos tópicos usualmente abordados en materias intermedias de las carreras de grado. Por ejemplo, en nuestro caso se trata de la asignatura *Organi ación de Computadoras* [2]. Esta materia forma parte del ciclo básico común a múltiples carreras, siendo en particular una de las asignaturas del segundo cuatrimestre de segundo año tanto de la *Licenciatura en Ciencias de la Computación* como en la *ngeniería en Sistemas de Computación*.

El rol principal de esta materia es servir de introducción al área de sistemas [11], abordando un conjunto de nociones básicas que luego serán retomadas y profundizadas en asignaturas posteriores tales como *Arquitectura de Computadoras*, *Sistemas Operativos*, *Sistemas Distribuidos* y *Redes de Computadoras*. Nuestra experiencia en el dictado de esta materia a lo largo del tiempo, señala que los alumnos encuentran diversas dificultades ajustándose al *bajo nivel de abstracción* con el cual se trabaja a medida que nos acercamos al hardware, máxime considerando que en el último cambio del plan de estudio de la licenciatura y de la ingeniería se optó por dar un tratamiento preferencial a la programación orientada a objetos, un paradigma que entre otras cosas se caracteriza por manejar un *alto nivel de abstracción*. En este sentido, nuestros alumnos dan sus primeros pasos como programadores en este paradigma, por lo que en ocasiones encuentran a la programación imperativa de bajo nivel, propia de las asignaturas del área de sistemas, un tanto contraria a las intuiciones que a esta altura de la carrera han comenzado a desarrollar. Atentos a esta objeción, identificamos que resultaría provechoso humanizar de alguna manera al hardware. En otras palabras, sería conveniente lograr que los alumnos sientan propio al hardware sobre el cual han de desarrollar la práctica asociada a esta asignatura. Una forma de alcanzar este objetivo consiste en hacer uso de una *arquitectura de juguete* en la cual el alumno esté a cargo de la totalidad de los procesos involucrados, desde la confección del programa, pasando por su compilación, vinculación y carga en memoria, hasta la propia ejecución del mismo. A tal efecto, en la cátedra hemos desarrollado una arquitectura de juguete denominada OCUNS (Organización de Computadoras UNS), cuya definción formal, aplicación desde un punto de vista pedagógico y metodológico e incluso implementación tentativa desarrollamos a continuación.

El resto de este artículo se organiza de la siguiente manera: la sección 2 define formalmente las características y el set de instrucciones de la arquitectura OCUNS. Luego, la sección 3 repasa y motiva el interesante rol que desde un punto de vista didáctico puede desempeñar esta arquitectura en el dictado de las asignaturas en las que se aborden estos tópicos introductorios al área de sistemas.

La sección 4 bosqueja el núcleo de una implementación tentativa en lenguaje JAVA para esta arquitectura. Finalmente, la sección 5 sintetiza las conclusiones alcanzandas durante el desarrollo del presente trabajo.

2. Arquitectura OCUNS

La arquitectura OCUNS está inspirada en parte en la arquitectura de juguete TOY desarrollada en la universidad de Princeton [3]. Consta de 256 bytes de memoria principal, instrucciones de 16 bits de tamaño fijo y 16 registros de propósito general. Se han contemplado sólo 16 tipos diferentes de instrucciones, las cuales se distinguen mediante los *opcodes* de 0 a F. Dada su naturaleza RISC [9], la arquitectura posee instrucciones que permiten manipular el contenido de los registros de propósito general, la memoria principal o bien el PC según se resume a continuación:

OPCODE	DESCRIPCIÓN	FORMATO	PSEUDOCÓDIGO
0	add (add)	**I**	`R[d] ← R[s] + R[t]`
1	sub (subtract)	**I**	`R[d] ← R[s] - R[t]`
2	and (and)	**I**	`R[d] ← R[s] & R[t]`
3	xor (xor)	**I**	`R[d] ← R[s] ^ R[t]`
4	lsh (left shift)	**I**	`R[d] ← R[s] << R[t]`
5	rsh (right shift)	**I**	`R[d] ← R[s] >> R[t]`
6	load (load)	**I**	`R[d] ← mem[offset + R[s]]`
7	store (store)	**I**	`mem[offset + R[d]] ← R[s]`
8	lda (load address)	**II**	`R[d] ← addr`
9	jz (jump zero)	**II**	`if (R[d] == 0) PC ← PC + addr`
A	jg (jump greater)	**II**	`if (R[d] > 0) PC ← PC + addr`
B	call (jump and link)	**II**	`R[d] ← PC; PC ← addr`
C	jmp (jump register)	**III**	`PC ← R[d]`
D	inc (increment)	**III**	`R[d] ← R[d] + 1`
E	dec (decrement)	**III**	`R[d] ← R[d] - 1`
F	hlt (halt)	**III**	`exit`

Figura 1. Set de instrucciones de la arquitectura OCUNS.

Estas 16 instrucciones se organizan en tres categorías, a saber:

- Instrucciones *aritmético-lógicas*.
- Instrucciones *de transferencia* entre la memoria principal y los registros.
- Instrucciones de *control de ujo*.

La codificación de cada una de las instrucciones consiste de cuatro dígitos hexadecimales (esto es, 16 bits). El primer dígito hexadecimal representa el *opcode* de la instrucción. El segundo dígito hexadecimal en la mayoría de los casos a uno de los 16 registros de propósito general, el cual denominaremos *registro destino*

y denotaremos como d. La interpretación de los últimos dos dígitos depende del *formato de instrucción* asociado. Para el primer formato de instrucción, tanto el tercero como el cuarto dígito hexadecimal se interpretan como sendos índices a registros de propósito general, denominados *registros origen* y denotados como s y t. Por caso, la instrucción **462** suma el contenido de los registros s = 6 y t = 2, colocando el resultado en el registro d = 4. Nótese que si bien se trata de una arquitectura RISC, hemos consevado la convención de la arquitecturas CISC de indicar típicamente en primer lugar el destino de la operación, para luego especificar el origen de los operandos. Para el segundo formato de instrucción, los últimos dos dígitos hexadecimales son interpretados como una *dirección de memoria*, la cual denotamos como addr. Por caso, la instrucción **462** almacena en el registro d = 4 la dirección de memoria addr = 0x62. Finalmente, en el caso del tercer formato de instrucción, los dos últimos dígitos hexadecimales no son requeridos (es decir, pueden contener cualquier valor).

A manera de síntesis, la siguiente tabla resume los distintos formatos de instrucción disponibles:

FORMATO	15 14	13 12 11 10	9 8 7 6 5 4	3 2 1 0
I	0	dest. d	src. s	src. t/off.
II	1 0	dest. d	address addr	
III	1 1	dest. d	-	

La arquitectura OCUNS representa a los enteros internamente en complemento a dos. Siguiendo el lineamiento de una arquitectura RISC, el último registro de propósito general (registro F) se encuentra *cableado a cero*, es decir, se comporta como si fuera un registro de sólo-lectura inicializado a cero.

3. Rol didáctico

La arquitectura OCUNS está siendo utilizada actualmente como herramienta para complementar la enseñanza de diversos conceptos en la asignatura *Organización de Computadoras*, con resultados satisfactorios. A continuación reseñaremos los distintos tópicos que resultan integrados al desarrollar una ejercitación práctica sobre esta arquitectura, prestando especial atención a las consideraciones metodológicas y didácticas.

En primer lugar, cabe aclarar que esta arquitectura permite desarrollar dos tipos de ejercicios:

- **ercicios de análisis:** el alumno a partir de un programa en código máquina debe estudiar su funcionamiento a fin de determinar cuál es su propósito, realizando una o más trazas de la evolución del estado de los registros en el tiempo.
- **ercicios de sín esis:** el alumno partiendo de la especificación de un problema, debe en primer lugar proponer una solución tentativa para luego bosquejar una implementación en lenguaje ensamblador. Opcionalmente, se puede solicitar al alumno que efectivamente ensamble el programa propuesto.

Ambos tipos de ejercicio puede ser intercalados ya que se complementan uno a otro. En cierto sentido, los ejercicios de análisis permiten ilustrar buenas prácticas de programación, en la idea de que el alumno las haga eventualmente propias en el desarrollo de los ejercicios de síntesis. Va de suyo que es recomendable comenzar por ejercicios de una complidad reducida, para luego ir incrementando la dificultad paulatinamente.

Conceptos tan importantes y centrales como la representación de números en distintas bases, la codificación de instrucciones y de parámetros, el formato de instrucción, el proceso de ensamblado, el proceso de vinculación, relocación y carga, resultan aplicados al desarrollar una ejercitación con la arquitectura OCUNS. Desde un punto de vista pedagógico, posibilitamos que el alumno complemente el *aprendi aje significativo* [4] que todo profesor intenta brindar en las clases teóricas con la posibilidad de experimentar un *aprendi aje por descubrimiento* [6] durante las clases prácticas.

Al tratarse de una arquitectura RISC, el set de instrucciones no cuenta con una instrucción de propósito específico para cada tarea que el programador requiera. Por caso, no se cuenta con una instrucción para inicializar a cero el valor de un registro. Resulta muy interesante invitar al alumno a descubrir cómo compensar esta aparente deficiencia, encontrándole un uso alternativo a alguna de las instrucciones de las que sí dispone la arquitectura. A su vez, resulta grátamente satisfactorio que el alumnado proponga nuevas e ingeniosas soluciones a este ingenuo planteo. Quizás resulte obvio restar consigo mismo el valor del registro que se desea inicializar (`sub Rx, Rx, Rx`) o bien sumar dos veces el registro cableado a cero (`add Rx, RF, RF`). No obstante, los alumnos también han encontrado alternativas un tanto más inusitadas como hacer uso de la instrucción `lda` para cargar la dirección `0x00`, o bien utilizar las instrucciones que realizan operaciones lógicas por nombrar solo algunas.

En lo que a la pila del sistema *(stack)* respecta, ese área de memoria extensamente utilizada para implementar el pasaje de parámetros en los distintos lenguajes de programación, la arquitectura OCUNS no cuenta con mecanismo alguno que realice la gestión de la misma de forma automática (por ejemplo, carece de *stack pointer*). En consecuencia, es posible desarrollar una práctica específica que involucre al alumno en la gestión de la pila del sistema. La idea es que la responsabilidad descance enteramente del lado del programador, que sea él quien destine uno de los registros de propósito general para hacer las veces de puntero al tope de pila (stack pointer) y que también reserve una porción de la memoria principal para alojar dicha estructura. En nuestro caso los ejercicios más avanzados desafían al alumno a implementar subrutinas, las cuales no afecten el estado previo de los registros de propósito general más allá de los utilizados para suministrar parámetros. De esta forma, el alumno puede ubicarse en una posición en la cual descubra por cuenta propia la complejidad asociada a preservar el estado del procesador. Motivado por esto se le aconseja implementar a nivel de software el funcionamiento de la pila del sistema, reservando uno de los registros de propósito general, por caso el registro `E`, para actuar como stack pointer, encomendándole lo utilice de manera consecuente, asumido que la pila

del sistema ha de comenzar en la locación 0xFF (incrementando su tamaño como es usual hacia direcciones decrecientes). En este sentido, los modos de direccionado *autoincremento* y *autodecremento*, cuyo rol en abstracto a veces resulta díficil de visualizar durante las clases teóricas (los alumnos frecuentemente los confunden entre sí), se transforman en características más que deseables a la hora de implementar a mano el funcionamiento de la pila del sistema, en particular luego de haber lidiado con las múltiples peculiaridades que se deben tener en cuenta al simularlos en la arquitectura OCUNS. Una vez que se tiene a disposición la pila del sistema se puede llegar incluso a abordar el desarrollo de programas recursivos. En nuestra experiencia, considerando en particular el acotado tiempo dedicado al desarrollo de la habilidad de construir programas en lenguajes de bajo nivel como el lenguaje ensamblador, se decidió restringir la práctica sobre programas recursivos a ejercicios de análisis, puesto que consideramos que la complejidad de los ejercicios de síntesis involucrando planteos recursivos estaba más allá del alcance de nuestra asignatura.

Otro aspecto que resulta bastante interesante de explorar con el objeto de poder escribir programas interactivos es la simulación de la entrada/salida. Esta tarea abarca avanzar sobre un área gobernada por el sistema operativo. Al hablar de una arquitectura de juguete esto implica que quien simule el funcionamiento del hardware (esto es, el alumno), también deberá simular el comportamiento del sistema operativo, al menos en lo que a esta funcionalidad respecta. Con el objeto de no complicar innecesariamente el desarrollo de los temas teóricos, ni de avanzar sobre conceptos que han de ser introducidos en materias posteriores, se optó por simular un mecanismo de entrada/salida bastante elemental, pero no por eso menos útil. La propuesta consiste en asumir que una determinada dirección de memoria presenta un comportamiento particular, de forma tal que todo acceso de lectura y/o escritura ha de generar un *trap* al sistema operativo. Toda vez que se almacene un cierto valor en esa locación de memoria, el sistema operativo durante la resolución del trap ha de mostrar por el dispositivo estándar de salida ese valor; caso contrario, intentar acceder al contenido de esa locación, el sistema operativo durante la resolución del trap ha de leer del dispositivo estándar de entrada un valor numérico el cual será considerado como el contenido de la locación de memoria accedida. Con esta simple suposición, es posible diseñar y simular la ejecución de programas interactivos no triviales sobre la arquitectura OCUNS.

En lo que respecta al proceso de ensamblado, vinculación y carga, tópicos centrales a la asignatura *Organi ación de Computadoras*, esta arquitectura permite que los alumnos posean una participación activa a lo largo de todas estas etapas. En contraste, el hacer uso de una arquitectura convencional (algo que los alumnos eventualmente también han de hacer una vez finalizada la práctica sobre la arquitectura OCUNS), estas tareas suelen quedar soslayadas por el entorno de desarrollo integrado (IDE) y también en cierta forma por la velocidad de los procesadores actuales, donde la compilación y vinculación de los programas confeccionados por los alumnos rara vez insume más de un par de milésimas de segundo. En primer lugar, el proceso de ensamblado permite al alumno explorar

el rol de los distintos formatos de instrucción disponibles, el funcionamiento de los distintos modos de direccionado (inmediato, registro directo y registro más desplazamiento), y el rol del registro PC. En nuestra experiencia, el proceso de ensamblado en sí ha sido de las actividades mas enriquecedoras dado que en dicho punto los alumnos logran dimensionar el funcionamiento básico de una computadora. En lo que al proceso de vinculación respecta, es posible explorar de qué manera se puede combinar el código objeto de distintos archivos fuente de forma tal de resolver correctamente las referencias involucradas. Finalmente, en cuanto al proceso de carga, es muy interesante ejercitar la relocación en memoria de código objeto ya compilado, invitando a los alumnos a analizar la totalidad del set de instrucciones, a la búsqueda de referencias absolutas las cuales requieran ser ajustadas. Particularmente, el mecanismo de invocación a subrutinas incluido en la arquitectura OCUNS (instrucciones call y jump) no hace uso de la pila del sistema para almacenar la dirección de retorno, sino que fuerza a la utilización de alguno de los registros de propósito general. Entendemos que la experimentación casi en primera persona de estos aspectos facilitan notablemente la comprensión de la implementación de los mismos adoptada en las arquitecturas convencionales.

Una característica propia del lenguaje ensamblador para la arquitectura OCUNS que amerita ser comentada, es la inclusión de etiquetas en el código para ser utilizadas como destino de las instrucciones de salto condicional (instrucciones jz y jg) e incondicional (instrucciones call y jump). En particular, la resolución de cómo se debe ensamblar la referencia a una dada etiqueta en los saltos condicionales (de naturaleza relativa) permite remarcar el rol del registro PC, capturando tempranamente el error típico de considerar que el contenido del mismo referencia a la dirección que actualmente está siendo ejecutada, puesto que el desplazamiento que se debe consignar en el campo addr de estas instrucciones se calcula como la diferencia entre el valor del PC y la dirección asociada a la etiqueta destino. La eventual aparición de un desplazamiento negativo (situación dada al implementar el salto hacia atrás de una estructura de control estilo while o repeat-until), posibilita poner en práctica los conocimientos adquiridos en relación a la representación de enteros signados, que en el caso de esta arquitectura recordemos se trata de la representación complemento a dos.

```
0x00:        lda RA, 0xFF      ; inicializa RA en 0xFF
0x02:        xor RB, RB, RB    ; inicializa RB en cero
0x04:  lazo: load RC, 0 (RA)   ; lee el primer valor por teclado
0x06:        add RB, RB, RC    ; lo acumula en RB
0x08:        jg RC, lazo       ; y repite hasta ingresar un cero
0x10:        store RB, 0 (RA)  ; muestra el valor acumulado en RB
0x12:        hlt               ; finaliza la ejecución
```

Figura 2. Ejemplo de un programa en lenguaje ensamblador para OCUNS.

Por último, a manera de ejemplificación de las distintas técnicas repasadas en esta sección, la figura 2 ilustra el código fuente en lenguaje ensamblador OCUNS de un programa que solicita el ingreso por teclado de una secuencia de valores terminada en cero y a continuación muestra por pantalla su sumatoria.

4. Implementación

Si bien el espíritu de la arquitectura OCUNS es que los alumnos sean los responsables de programar, ensamblar, vincular, cargar y ejecutar sus propios programas, en ocasiones resulta conveniente contar con un emulador para la arquitectura. Por caso, al proponer ejercicios de análisis, ya sea en el marco de un trabajo práctico o bien de un examen parcial, es conveniente contar con la posibilidad de verificar que el programa propuesto se comporta de la manera anticipada. En este sentido, en esta sección reseñaremos los aspectos más relevantes de una implementación en lenguaje JAVA de un emulador para la arquitectura OCUNS que fuera realizada por la propia cátedra de la materia.

El hecho de que la arquitectura OCUNS se trate de una aquitectura RISC, simplifica notablemente la tarea de implementar un emulador para la misma. En primer lugar, las estructuras de datos para representar la memoria principal y el banco de registros de propósito general son meros arreglos de enteros y el registro de propósito específico PC es simplemente una variable:

```
int pc    = 0x00;         // program counter
int[] R   = new int[16];  // banco de registros
int[] mem = new int[256]; // memoria principal
```

Luego, el núcleo del emulador consiste en simular el ciclo de ejecución de toda arquitectura convencional: *fetch*, *decode* y *execute*. Para la etapa de fetch, simplemente se recompone una palabra de 16 bits a partir de las dos locaciones consecutivas de memoria referenciadas por el registro PC. Para la etapa de decode, básicamente se extraen los distintos campos que componen una instrucción haciendo uso de las operaciones de corrimiento de bits. Finalmente, para la etapa de execute, se hace uso del propio pseudo-código consignado en la tabla que introduce el set de instrucciones de la arquitectura (tabla 1). La figura 3 ilustra el ciclo infinito que constituye el núcleo del emulador.

5. Conclusiones

El dictado de las clases teóricas y prácticas en las asignaturas del área de sistemas puede acarrear desafíos particulares y específicos, producto de la separación entre el alto nivel de abstracción que usualmente se maneja en el resto de las materias de grado en ciencias de la computación, y el bajo nivel inherente a los conceptos cercanos al hardware que constituyen la columna vertebral de las nociones más avanzadas de este área. En este sentido, a lo largo del presente

```
int inst, op, d, s, t, addr, offset;

while(true) { // lazo eterno de fetch, decode y execute

   // etapa de fetch
   lowb   = mem[pc];
   highb  = mem[pc + 1];
   inst   = lowb + (highb << 8);
   pc     = pc + 2;

   // etapa de decode
   op     = (inst >> 12) &  15;    // decode opcode (bits 12-15)
   d      = (inst >>  8) &  15;    // decode dest   (bits  8-11)
   s      = (inst >>  4) &  15;    // decode s      (bits  4- 7)
   t      = (inst >>  0) &  15;    // decode t      (bits  0- 3)
   addr   = (inst >>  0) & 255;    // decode addr   (bits  0- 7)
   offset = (t >= 8) ? t - 16 : t; // decode offset (bits  0- 7)

   // etapa de execute
   if (op == 0xF) break;

   switch (op) {
      case 0x0: R[d] = R[s] +  R[t];
                break; // add
      case 0x1: R[d] = R[s] -  R[t];
                 break; // subtract
      case 0x2: R[d] = R[s] &  R[t];
                break; // bitwise and
      case 0x3: R[d] = R[s] ^  R[t];
                break; // bitwise xor
      case 0x4: R[d] = R[s] << R[t];
                break; // shift left
      case 0x5: R[d] = (short) R[s] >> R[t];
                break; // shift right
      case 0x6: R[d] = mem[(R[s] + offset + 256) & 255];
                break; // load
      case 0x7: mem[(R[d] + offset + 256) & 255] = R[s];
                break; // store
      case 0x8: R[d] = addr;
                break; // load address
      case 0x9: if ((short) R[d] == 0) pc = pc + addr;
                break; // branch if zero
      case 0xA: if ((short) R[d] >  0) pc = pc + addr;
                break; // branch if positive
      case 0xB: R[d] = pc; pc = addr;
                break; // call
      case 0xC: pc = R[d];
                break; // jump
      case 0xD: R[d]++;
                break; // increment
      case 0xE: R[d]--;
                break; // decrement
   }

   R[15] = 0;          // el registro RF est\'a cableado a 0
   R[d] = R[d] & 0xFF; // wrap around del registro destino
   pc = pc & 255;      // wrap around del registro PC

}
```

Figura 3. Núcleo del emulador en JAVA para la arquitectura OCUNS.

artículo hemos caracterizado formalmente una arquitectura de juguete denominada OCUNS, la cual facilita la práctica por parte de los alumnos de diversos tópicos del área de sistemas. También hemos repasado brevemente los beneficios desde un punto de vista tanto pedagógico como metodológico, de hacer uso de este tipo de herramienta en las actividades prácticas de los alumnos. En este sentido, hemos remarcado en base a nuestra experiencia los aspectos mas relevantes de la utilización de la arquitectura por parte del alumnado. Por último, hemos bosquejado someramente el núcleo de una implementación en el lenguaje JAVA de esta arquitectura.

En lo que a trabajos a futuro respecta, estamos contemplando la elaboración de un enunciado de tesina o proyecto final de carrera (en el caso de la Ingeniería) para que los alumnos pongan en práctica los conocimientos adquiridos a lo largo de la carrera, diseñando un compilador para el lenguaje ensamblador de esta arquitectura que genere como salida código objeto. Cabe acotar que el emulador antes comentado recibe como entrada código objeto, es decir, actualmente no acepta como entrada código fuente en lenguaje ensamblador. La idea de este enunciado es extender la funcionalidad del emulador para que acepte directamente código fuente en lenguaje ensamblador para la arquitectura OCUNS. Una vez que este proyecto esté concluido, es nuestra intención licenciar su código fuente bajo alguna licencia de software libre para alentar su uso y difusión.

Referencias

1. Departamento de ciencias e ingeniería de la computación. `http://cs.uns.edu.ar`
2. Organización de computadoras. `http://cs.uns.edu.ar/~ags/OC`
3. Toy architecture. `http://introcs.cs.princeton.edu/java/52toy/`
4. Ausubel, D.P.: A subsumption theory of meaningul velbal learning and retention. Journal of General Pshychology 66, 213–224 (1962)
5. Ausubel, D.P., Novak, J.D., Hanesian, H.: Educational Psychology: A Cognitive View. Holt McDougal, 2nd edn. (1978)
6. Bruner, J.S.: Towards a Theory of Instruction. Harvard University Press (1966)
7. Hennessy, J., Patterson, D.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann, 4th edn. (2006)
8. Patterson, D., Hennessy, J.: Computer Organization Design: The Hardware/Software Interface. Morgan Kaufmann, 3rd edn. (2007)
9. Patterson, D.A., Ditzel, D.R.: The case for the reduced instruction set computer. SIGARCH Comput. Archit. News 8, 25–33 (October 1980), `http://doi.acm.org/10.1145/641914.641917`
10. Stallings, W.: Computer Organization and Architecture. Prentice Hall, 8th edn. (2009)
11. Tanenbaum, A.S.: Structured Computer Organization. Prentice Hall, 5th edn. (2005)